CPSC 311: **Practice** Midterm Exam #1 Solution

October 2, 2018

We **CANNOT EMPHASIZE ENOUGH** that looking at sample solutions without putting significant effort into working the blank exam first **and** discussing/critiquing your solutions with someone is not generally a good study strategy.

1 I'm Syntax, Your Nemesis [?? marks]

Here is the original syntax for the EP language, expressed in EBNF:
SOLUTION: inline

```
<Expr> ::= <num> | <id>

         | {<binop> <Expr> <Expr>}

         | {fun {<id>} <Expr>}

         | {with {<binding>*} <Expr>}

         | {extend <Expr> <mapping>}

         | {<Expr> <Expr>}              ; function application


<binop> ::= + | - | * | /


<binding> ::= {<id> <Expr>}
            | {def {<id> <id>} <Expr>}

<mapping> ::= {<Expr> => <Expr>}
```


1. Neatly edit the EBNF above to add a new way to bind identifiers inside of a `with` expression in order to conveniently define a function. **[?? marks]**

   This should look like:

   ```
   {with {{def {double n} {+ n n}}}
     {double 5}}
   ```

   which would be equivalent to:

   ```
   {with {{double {fun {n} {+ n n}}}}
     {double 5}}
   ```

   Note that it should be possible to freely mix original- and function-style bindings in a single `with` expression, e.g.:

   ```
   {with {{x 5}
          {def {add-x y} {+ x y}}}
     {add-x 10}}
   ```

   SOLUTION: Inline above.

2. Select the easier strategy of (1) implementing this as a desugaring or (2) introducing a new `withfun` AST node and implementing this new binding type in interpretation, and then complete the implementation. Argue for why your strategy is easier and for whether it is *better*. **[?? marks]**

   SOLUTION: We think a desugaring would be easiest. Here's a parser case, with the existing 1+ binding `with` followed by our new one:

```
(define (parse sexp)
  (local [(define binops ...)]
    (match sexp
      ...

      ; All other withs:
      [(list 'with (cons (list (? valid-id? name) named-exp) bindings) body)
       ; Here's the desugaring. Effectively:
       #;{{fun {<name>} {with <bindings> <body>}} <named-exp>}
       (parse `{{fun {,name} {with ,bindings ,body}} ,named-exp})
       #;(app (fun name (parse (list 'with bindings body)))
            (parse named-exp))]

      ; We'll desugar to a "normal-style" binding via recursive parsing:
      [(list 'with (cons (list 'def (list fname pname) fbody) bindings) body)
       (parse `{with {{,fname {fun {,pname} ,fbody}}}
                 {with ,bindings ,body}})]

      ...

      [_ (error 'parse "Something went wrong in ~a" sexp)])))
```

We think this strategy is the easier one since it requires little to no change in the overall parser/interpreter, and the desugaring of the binding `{def {<f> <p>} <body>}` to the binding `{<f> {fun {<p>} <body>}}` is fairly straightforward.

Whether this strategy is better may depend on some larger issues. For example, the desugaring loses some information present in the original program, such as dissociating the function name from the (now-anonymous) function at the moment the function value is created. The interpreter solution maintains this information, but note that **our** interpreter has already lost information that's important to the end-user programmer, such as the original line number on which this binding occurred. To use our system in production, we'd want to determine the information we want to maintain throughout the system and record it. At that point, it would probably be doable to maintain aspects like the function's name across our desugaring. So.. it depends :)

2 Dynaguise [?? marks]

An `extendClosureV` is a variant of `Value` defined as:

```
[extendClosureV (key Value?) (rexp Expr?) (enclosed-fun Value?)] ; CUT the env field
```

It stores the unevaluated result expression (`rexp`) and the environment (`env`) in which the `extend` expression was evaluated. We use this code to apply an `extendClosureV`:

```
[extendClosureV (kval rexp fval)  ; CUT cenv binding
  (if (and (numV? kval) (numV? aval) (= (numV-n kval) (numV-n aval)))
    (helper rexp env)          ; <-- CHANGED from cenv to env
    (apply fval aval) env)]))  ; <-- ADDED env as a parameter to recursive call
```

Consider **two** approaches to that result expression:

1. The **original** approach above.
2. The **body only** approach: We discard the environment included in the `extendClosureV`, using the current environment of the expression applying the `extendClosureV` instead.

Now, answer the questions below.

1. Neatly edit the value type and interpreter above as needed to implement the **body only** strategy.

   SOLUTION: inline above. We removed the environment field from `extendClosureV` and altered the call `(helper rexp cenv)` to `(helper rexp env)`. Note, however, that we also need `env` as a parameter, which means it needs to come from somewhere! (It's in scope within `apply`, but the one we have access to is the `env` from the outermost call to `interp-env`, **not** the local environment at the application point.) That's the more general change we need to make...

   In addition, clearly explain the one other change (or set of changes) required in the broader interpreter to implement these semantics. **[?? marks]**

   SOLUTION: We need to change `apply` to take an environment parameter. At each call site, we simply pass the locally available environment (which happens to always be named `env`).

2. Does the **body only** approach introduce dynamic scoping? Fill in the circle next to the **best** answer. **[?? marks]**

   - ● Yes, because the environment used is not the static environment of the `extend` expression.
   - ○ Yes, because the parameter leaks dynamic scope into the result expression
   - ○ No, because dynamic scoping requires function **definition** not just function application.
   - ○ No, because mappings cannot directly access the extended function's parameter.
   - ○ It's not possible to tell with the details given in this problem.

   SOLUTION: Yep. Dynamic scoping is when we allow the environment to pass along the dynamic "flow" of the program, even when it "jumps around" in the static structure of the program, which is exactly what we're doing here. You can see that in the rewriting to pass plain old `env` into function application.

   It doesn't change **all** scoping to be dynamic in our program, but it certainly **introduces** dynamic scoping.

3. Each of the following programs in EP has one or two boxes after it for you to fill in the value of the program. Each box is labelled with semantics to use in evaluation (**original** or **body only**). Write in each box the value of the program under the requested semantics. **[?? marks]**

(a) `{+ 1 4}`

Body only: $\boxed{5}$

(b) `{fun {x} y}`

Original: $\boxed{\text{other}}$

SOLUTION: A closure like `(closureV 'x (id 'y) (mtEnv))`.

(c)
```
{{extend {fun {x} {+ x 10}}
         {y => 100}}  ; <-- unbound in the KEY expression
 {with {{y 3}} y}}
```

Original: $\boxed{\text{error}}$ **Body only:** $\boxed{\text{error}}$

SOLUTION: The key expression is evaluated immediately in both variants; so, this causes an error in both versions.

(d)
```
{with {{x 100}
       {f {extend {fun {x} {+ x 10}}
                  {x => y}}}}
  {with {{x 1} {y 1000}}
    {f 1}}}
```


Original: $\boxed{11}$ **Body only:** $\boxed{11}$

SOLUTION: This may seem surprising. If we defer evaluation of the **key** expression until its use *and* scope it dynamically, we'd find the binding $x \mapsto 1$, match the function argument `1` against that, and return 1000 for the (dynamically-scoped) value of y.

However, as mentioned above, the key expression is eagerly evaluated. So, both versions' mappings are from `100`. Thus, both versions return the result of the enclosed function when called on `1`, which is `11`.

(e)
```
{with {{x 100}
       {f {extend {fun {x} {+ x 10}}
                  {x => y}}}}
  {with {{x 1} {y 1000}}
    {f 100}}}
```

Original: $\boxed{\text{error}}$ **Body only:** $\boxed{1000}$

SOLUTION: And here's where all that reasoning above pays off. When we call `{f 100}`, we get 1000 for the body only version but an error on the unbound identifier `y` for the original version.

(f)
```
{{extend {fun {x} 100}
         {{fun {y} y} => 10}}
 {fun {z} z}}
```

Original: $\boxed{100}$

SOLUTION: The key expression is supposed to have a numeric value. There's a surprising bit in the specification that says that key expressions of the wrong type are essentially ignored rather than creating an error, which **is** how they are implemented. That spec should perhaps change, but as is, it means that the closure argument passed in bypasses the mapping (even though one might argue that these are semantically identical functions) and gives the enclosed function's result.

Aside: It turns out to be impossible in the general case to compare functions for equality; so, at least disallowing them in the key position is probably a good idea, though creating an error might be an even better idea!

(g)
```
{with {{g {with {{f {fun {x} {+ x 10}}}}
            {extend f {{f 1} => {f 10}}}}}}
  {g {g 1}}}
```

Original: 20 **Body only:** error

SOLUTION: In the original approach, we map `{f 1} = 11` to `{f 10} = 20`. `{g 1}` is the same as `{f 1}`. So, `{g {g 1}} = {g {f 1}} = {g 11}`, which is `{f 10}`, but in the body only approach, this is evaluated **outside** the `with` that binds `f`; so, we get an unbound identifier error when we try to evaluate `{f 10}`.

(h)
```
{with {{f {fun {x} {+ x 10}}}
       {g {extend f {{f 1} => {g 10}}}}}
  {g {g 1}}}
```


Body only: 20

SOLUTION: Dynamic scoping makes implementing recursion easy!

`g` **is** available in the body of the `with`; so, the call to `{g 10}` behaves normally. `10` is not overridden in the `extend`; so, we get `{f 10} = 20`.

(i)
```
{with {{x 0}
       {f {extend {fun {x} x}
                  {1 => {/ 1 x}}}}
       {x 1}}
  {f 2}}
```

Original: 2 **Body only:** 2

SOLUTION: 2 isn't remapped in the extension; so, we get the identity function `{fun {x} x}` applied to 2, which is just 2.

(j)
```
{with {{x 0}
       {f {extend {fun {x} x}
                  {1 => {/ 1 x}}}}
       {x 1}}
  {f 1}}
```

Original: error **Body only:** 1

SOLUTION: Now dynamic scoping kicks in again, and we get the second binding of `x` under the **body only** approach but the first under the **original** approach.

3 The Undefiner [?? marks]

We want to add three new forms to our language:

```
<Expr> ::= {undefine <id> <Expr>}
         | {unmap <Expr> {<Expr> => ..}}
         | {withdef <id> <Expr>}
```

`Value` has been edited to add the variant `[undefV]`. Evaluating a `{undefine <id> <Expr>}` form causes `<id>` to be bound to `(undefV)` with a static scope of the `<Expr>`, and then returns the result of evaluating the `<Expr>`. To evaluate a `{unmap <Expr> {<Expr> => ..}}` form, we evaluate the first `<Expr>` to a function value f (a `closureV` or `extendClosureV`), evaluate the second `<Expr>` to a numeric value n, and then produce a function value that extends f such that if it is applied to n, it produces `(undefV)` and otherwise it applies f to the argument value and returns the result. A `withdef` evaluates to `(undefV)` if `<id>` is bound to `(undefV)` and otherwise evaluates `<Expr>` and returns its value. (It produces an error if `<id>` is unbound.)

1. Here are the parser cases for `extend` and a new (correct) one for `undefine`. **Neatly add** a new parser case for `{unmap <Expr> {<Expr> => ..}}` that is implemented by desugaring it using the new `{undefine <id> <Expr>}` form and the previously existing ones. (Note that `..` is just syntax constisting of two consecutive periods and not some special EBNF symbol.) **[?? marks]**

   SOLUTION: inline below along with explanation


```
(define (parse sexp)
  (local [(define binops ...)]
    (match sexp ...
      [(list 'extend fun-expr (list key-expr '=> res-expr))
       (extend (parse fun-expr) (parse key-expr) (parse res-expr))]
      [(list 'undefine (? valid-id? id) body-expr)
       (undefine id (parse body-expr))]
      ; SAMPLE SOLUTION, with match code based mostly off the extend case above.
      [(list 'unmap fun-expr (list key-expr '=> '..))
       ; If we can create an (undefV), the desugaring becomes easy.
       ; Look at {undefine x ...}. Inside the ..., x is bound to (undefV).
       ; So, we just return x: {undefine x x}
       (parse `{extend ,fun-expr {,key-expr => {undefine x x}}})]
       ; THOUGHT EXPERIMENT: Is x OK to use as a name? Will it cause problems?

      [_ (error 'parse "Something went wrong in ~a" sexp)])))
```


2. Here is the existing datatype used for the language's abstract syntax. **Neatly complete** the new variants named `undefine` and `withdef` to represent the `undefine` and `withdef` concrete syntax. **[?? marks]**

   SOLUTION: inline with explanation below

```
(define-type Expr
  [num (n number?)]
  [id (name symbol?)]
  [binop (op procedure?) (lhs Expr?) (rhs Expr?)]
  [fun (param symbol?) (body Expr?)]
  [extend (fun-exp Expr?) (key-exp Expr?) (result-exp Expr?)]
  [app (fun-exp Expr?) (arg-exp Expr?)]
```

```
[undefine (name symbol?) (body Expr?)]

[withdef (name symbol?) (body Expr?)]
```

SOLUTION: Inline above. Sadly, they both have essentially the same form! Their form is the same as a `fun` and similar to a `with`. The fields needn't be named `name` and `body`, but they should be reasonable names and should have exactly the given types.

3. Neatly edit the interpreter to implement the desired behaviour for `undefine` and `withdef`. **[?? marks]**

SOLUTION: inline below

```
(define (interp-env exp env)
  (local [(define (apply fval aval)
            (type-case Value fval
              [numV (n) (error 'interp-env "attempt to apply a number ~a as a function in ~a"
                               n exp)]
              [closureV (param body cenv)
                        (helper body (anEnv param aval cenv))]
              [extendClosureV (kval rexp cenv fval) ; <-- shadowing fval, which is OK
                (if (and (numV? kval) (numV? aval) (= (numV-n kval) (numV-n aval)))
                  (helper rexp cenv)
                  (apply fval aval))]))

          ...

          (define (helper exp env)
            (type-case Expr exp
              ...
              ; Insert your cases here.
              [undefine ( name body)
                ; Like a with, but we already know the value of the named expression:
                ; it's (undefV).
                (helper body (anEnv name (undefV) env))

                ]

              [withdef ( name body     )
                ; A bit like an if/else combined with an id reference:
                (local [(define val (lookup-env name env))]
                  (if (undefV? val)
                      val
                      (helper body env)))
                  ]

              [extend (fexp kexp rexp)
                      (extendClosureV
                       (assert-type (helper kexp env) numV?) rexp env
                       (assert-type (helper fexp env) (or/c closureV? extendClosureV?)))]
              [app (fexp aexp)
                   (apply (helper fexp env) (helper aexp env))]))]
    (helper exp env)))
```

4 Mint Condition in its Original Set-Box [?? marks]

Imagine we have correctly added mutation via boxes to our language, with these extra syntactic forms using their usual semantics (where `set-box!` returns the new value placed in the box):

```
<Expr> ::= {box <Expr>}
         | {set-box! <Expr> <Expr>}
         | {unbox <Expr>}
```

This requires defining an order of evaluation of subexpressions. Where that is not already explicit in the language semantics, it will be left-to-right within an expression.

In each box after a program below, write the result of evaluating the program as a number (e.g., for the program `{+ 1 2}`, write 3). **All results** are numbers. **[?? marks per problem]**

1. `{unbox {box 3}}`

   3

2.
```
{with {{b {box 1}}
       {f {fun {x} {set-box! x {* 10 {unbox x}}}}}
       {g {extend f {1 => {set-box! b {* 2 {unbox b}}}}}}}
  {+ {g b}
     {g 1}}}
```

   30

   SOLUTION: `f` multiplies the value in its box parameter by 10. `g` extends `f` so that if it receives a (numeric) 1, it doubles the value in `b`. Then we add the result of multiplying b's contents by 10 (10, and now b contains 10) and then doubling b's contents (20), which is 30.

3.
```
{with {{f  {fun {x} {box 1}}}  ; <-- returns a (fresh) box [1]
       {b1 {f 1}}              ; <-- b1 is one box [1]
       {b2 {f 1}}}             ; <-- b2 is a DIFFERENT box [1]
  {+ {set-box! b1 {* 10 {unbox b1}}}   ; <-- b1 = [10]
     {set-box! b2 {* 10 {unbox b2}}}}} ; <-- b2 = [10]
```

   20

   SOLUTION: See the notes above. Critically, each time we call `f`, we produce a **new** box, not a new reference to the same box.

4.
```
{with {{f  {fun {x} {box 1}}}          ; <-- return a fresh box
       {g  {extend f {1 => {box 1}}}}  ; <-- on 1 return a fresh box
       {b1 {g 1}}                      ;
       {b2 {g 1}}}                     ; As a result, the rest of this
  {+ {set-box! b1 {* 10 {unbox b1}}}   ; is much like the previous problem!
     {set-box! b2 {* 10 {unbox b2}}}}};
```

   20

   SOLUTION: See above in comments.

5. ```
{with {{f {fun {x} x}}
       {g {extend f {2 => {box 10}}}}
       {b {box 1}}}
  {+ {set-box! b 2}
     {unbox {g b}}}}
```

$\boxed{4}$

**SOLUTION:** `f` is the identity function. `g` is the same as `f` except with the parameter 2, it returns a fresh box containing 10.

It's somewhat unclear exactly what our extended closures do in the presence of boxes. If we leave their code as it is now, they simply don't match keys that are boxes. Then, this will set the value in `b` to 2 and then unbox that same value of 2 again, adding them to 4.

Conceivably, you might decide that the implementation **does** match on boxes if their contents are numbers (though in that case, you should think about what happens when the contents of the box are… a box). In that case, this actually returs 12 instead, which we would also accept as a reasonable answer.
```